System design: Typeahead Suggestion System

Suresh Podeti
8 min read · Oct 25, 2023

Introduction

Typeahead suggestion, also referred to as the **autocomplete feature**, enables users to search for a known and frequently searched query. This feature comes into play when a user types a query in the search box. The typeahead system provides a list of suggestions to complete a query based on the user's search history, the current context of the search, and trending content across different users and regions. Frequently searched queries always appear at the top of the suggestion list. **The typeahead system doesn't make the search faster**. However, **it helps the user form a sentence more quickly**. It's an essential part of all search engines that **enhances the user experience**.

Requirements

Functional

The system should suggest **top *N*** (let's say top ten) **frequent** and **relevant terms** to the user based on the text a user types in the search box.

Non-functional

- **Low latency:** The system should show all the suggested queries in real time after a user types. The latency shouldn't exceed 200 ms. A study suggests that the average time between two keystrokes is 160 milliseconds. So, our time-budget of suggestions should be greater than 160 ms to give a real-time response. This is because if a user is typing fast, they already know what to search and might not need suggestions. At the same time, our system response should be greater than 160 ms. However, it should not be too high because in that case, a suggestion might be stale and less useful.

- **Fault tolerance:** The system should be reliable enough to provide suggestions despite the failure of one or more of its components.
- **Scalability:** The system should support the ever-increasing number of users over time.

High-level design

System shouldn't just **suggest queries** in real time with minimum latency but should also **store the new search queries** in the database. This way, the user gets suggestions based on popular and recent searches.

When a user starts typing a query, every typed character hits one of the application servers. Let's assume that we have a **suggestions service** that obtains the top ten suggestions from the cache, **Redis**, and returns them as a response to the client. In addition to this, suppose we have another service known as an **assembler**. An assembler collects the user searches, applies some analytics to rank the searches, and stores them in a **NoSQL database** that's distributed across several nodes.

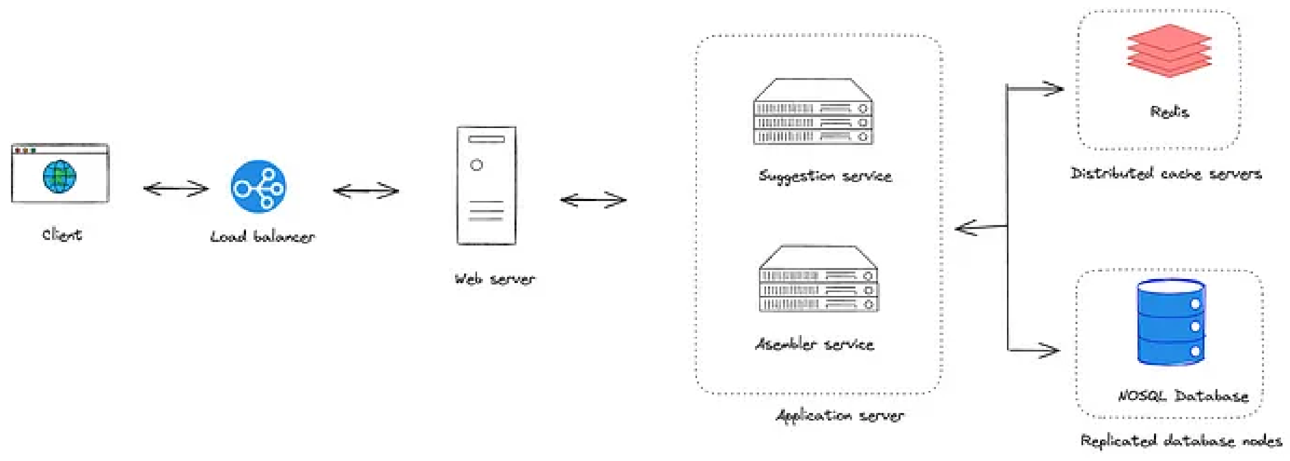


Fig 1.0: High level design of typeahead suggestion system

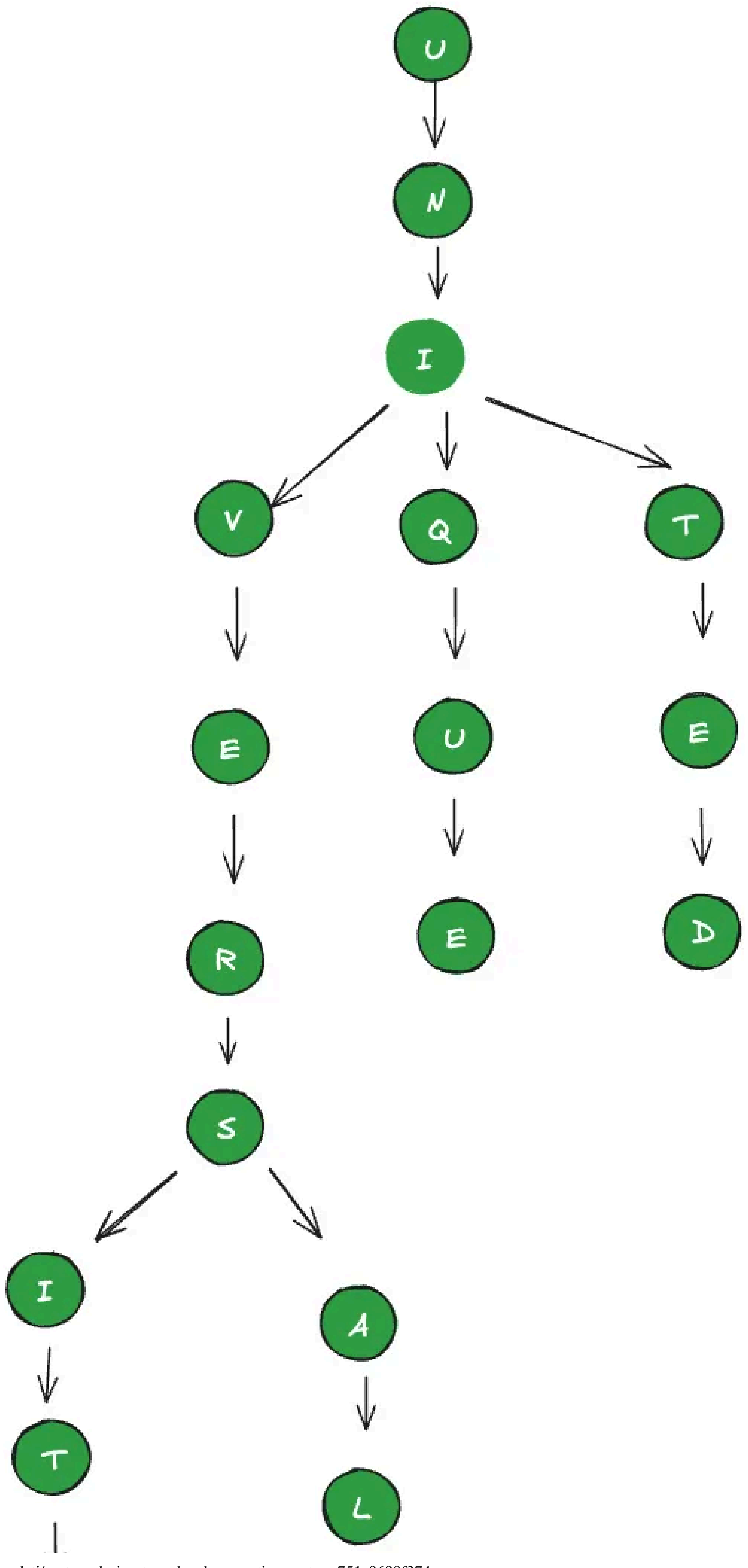
Data structure

The trie data structure

We must choose an efficient data structure to store the **prefixes.** Prefixes are the groups of characters a user types. We need to store in a way that allows users to search for them using any prefix. Let's suppose our database contains the phrases `UNITED`, `UNIQUE`, `UNIVERSAL`, and `UNIVERSITY`. Our system should suggest "**UNIVERSAL**" and "**UNIVERSITY**" when the user types "**UNIV.**"

There should be a method that can efficiently store our data and help us conduct fast searches because we have to handle a lot of requests with minimal latency. We can't rely on a database for this because providing suggestions from the database takes longer as compared to reading suggestions from the **RAM**. Therefore, we need to store our index in memory in an efficient data structure. However, for durability and availability, this data is **stored in the database**.

The **trie** (pronounced "try") is one of the data structures that's best suited to our needs. A **trie** is a tree-like data structure for storing phrases, with each tree node storing a character in the phrase in order. If we needed to store `UNITED`, `UNIQUE`, `UNIVERSAL`, and `UNIVERSITY` in the trie, it would look like this:


U
N
I
V
Q
T
E
U
E
R
E
D
S
I
A
T
L

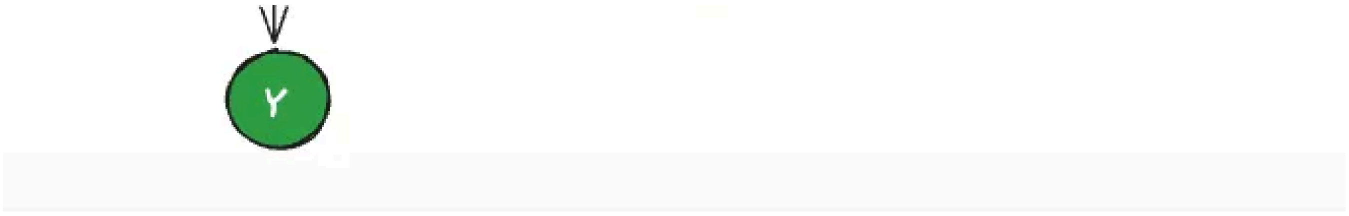


Fig 2.0: The trie for UNITED, UNIQUE, UNIVERSAL, and UNIVERSITY

If the user types "UNIV," our service can traverse the trie to go to the node `V` to find all the terms that start with this prefix—for example, `UNIVERSAL`, `UNIVERSITY`, and so on.

The trie can **combine** nodes as one where only **a single branch** exists, which **reduces the depth of the tree**. This also reduces the traversal time, which in turn increases the efficiency. As an example, a space- and time-efficient model of the above trie is the following:

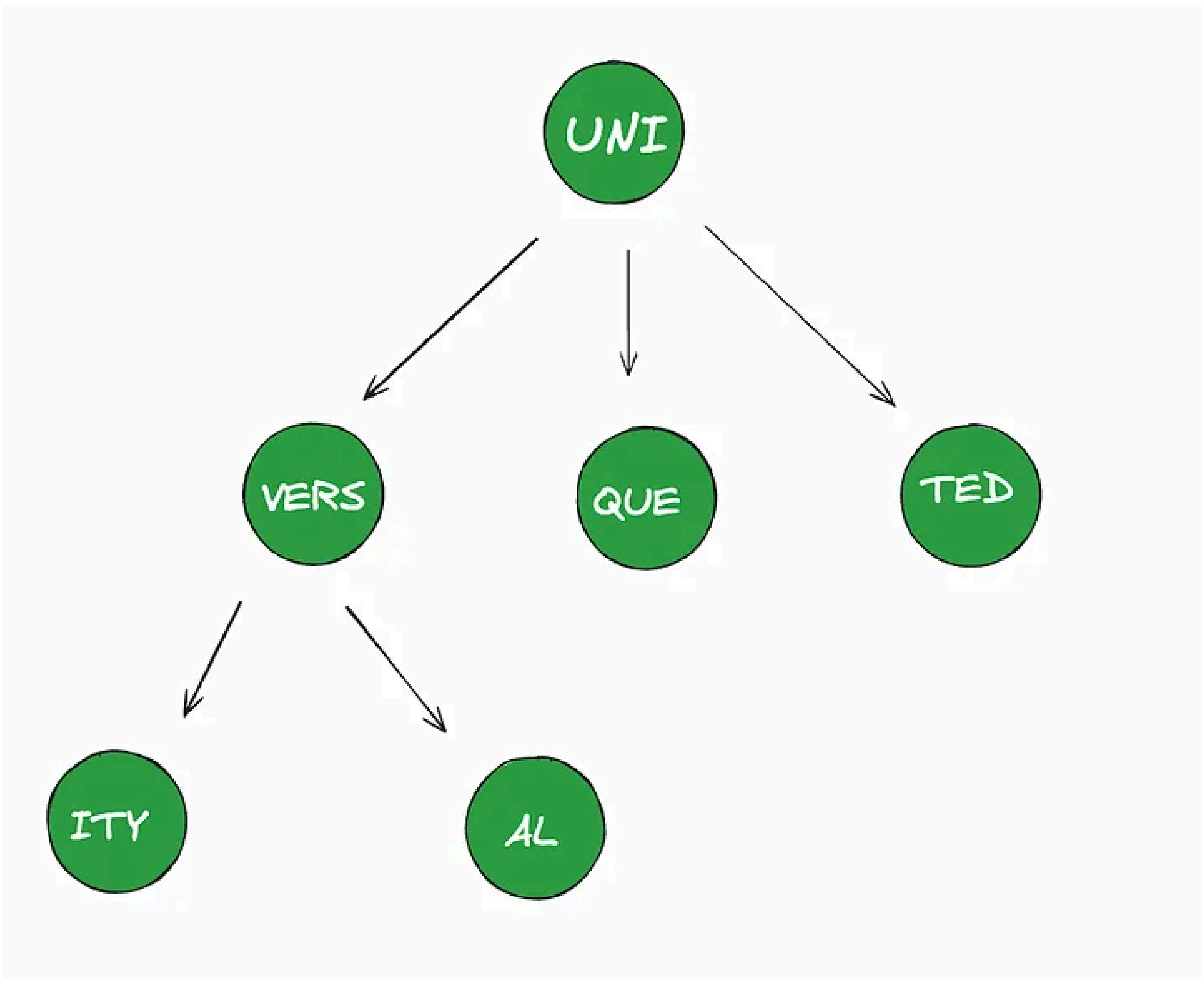


Fig 3.0: A reduced Trie for UNITED, UNIQUE, UNIVERSAL, and UNIVERSITY

Track the top searches

Since our system keeps track of the top searches and returns the top suggestion, we store the number of times each term is searched in the trie node. Let's say that a user searches for `UNITED` 15 times, `UNIQUE` 20 times, `UNIVERSAL` 21 times, and `UNIVERSITY` 25 times. In order to provide the top suggestions to the user, these counts are stored in each node where these terms terminate. The resultant trie looks like this:

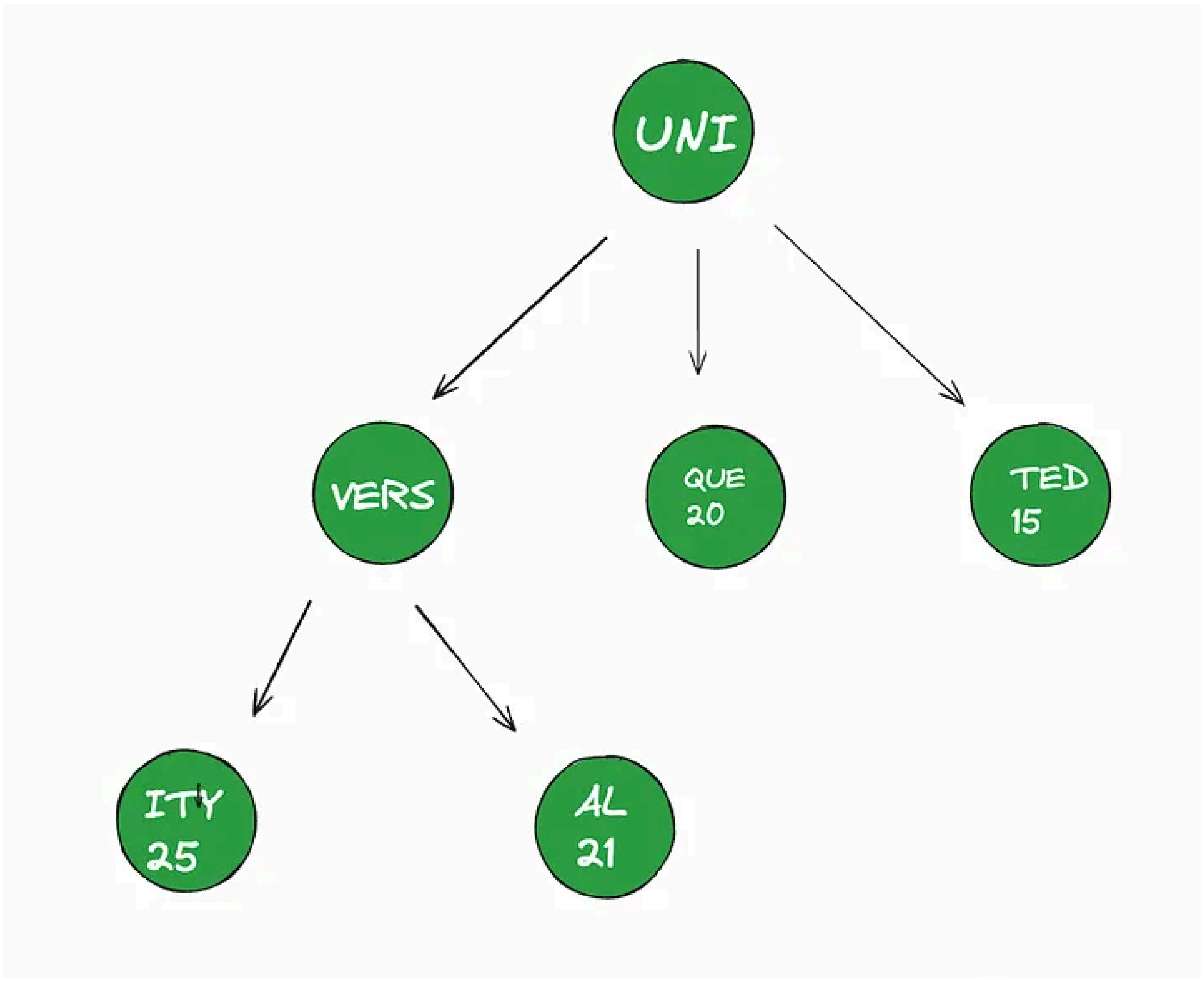


Fig 4.0: A trie showing the search frequency for UNITED, UNIQUE, UNIVERSAL, and UNIVERSITY

If a user types "UNI," the system starts traversing the tree under the root node for `UNI` . After comparing all the terms originating from the root node, the system provides suggestions of all the possible words. Since the frequency of the word `UNIVERSITY` is high, it appears at the top. Similarly, the frequency of the word `UNITED` is relatively low, so it appears last. If the user picks `UNIQUE` from the list of suggestions, the number against `UNIQUE` increases to 21.

Trie partitioning

We aim to design a system like Google that we can use to handle billions of queries every second. One server isn't sufficient to handle such an enormous amount of requests. In addition to this, storing all the prefixes in a single trie isn't a viable option for the system's availability, scalability, and durability. A good solution is to **split the trie into multiple tries** for a better user experience.

Let's assume that the trie is split into four parts, and each part has a replica for durability purposes. All the prefixes starting from "A" to "M" are stored on Server/01, and the replica is stored on Server/02. Similarly, all the prefixes starting from "N" to "Z" are stored on Server/03, and the replica is stored on Server/04. It should be noted that this simple technique doesn't always balance the load equally because some prefixes have many more words while others have fewer.

Process a query after partitioning

When a user types a query, it hits the **load balancer** and is forwarded to one of the application servers. The application server searches the appropriate trie depending on the prefix typed by the user.

Update the trie

Billions of searches every day give us hundreds of thousands of queries per second. Therefore, the process of updating a trie for every query is highly resource intensive and time-consuming and **could hamper our read requests**. This issue can be resolved by updating the trie **offline** after a **specific interval**. To update the trie offline, we log the queries and their frequency in a hash table and aggregate the data at regular intervals. After a specific amount of time, the trie is updated with the aggregated information. After the update of the trie, all the previous entries are deleted from the hash table.

We can put up a **MapReduce (MR)** job to process all of the logging data regularly, let's say every 15 minutes. These MR services calculate the frequency of all the searched phrases in the previous 15 minutes and dump the results into a hash table in a database like Cassandra. After that, we may further update the trie with the new data. We can update the current copy of the trie with all of the new words and their frequencies. We should perform this offline because our priority is to provide suggestions to users instead of keeping them waiting.

Another way is to have one primary copy and several secondary copies of the trie. While the main copy is used to answer the queries, we may update the secondary copy. We may also make the secondary our main copy once the upgrade is complete. We can then upgrade our previous primary, which will then be able to serve the traffic as well.

Detailed design

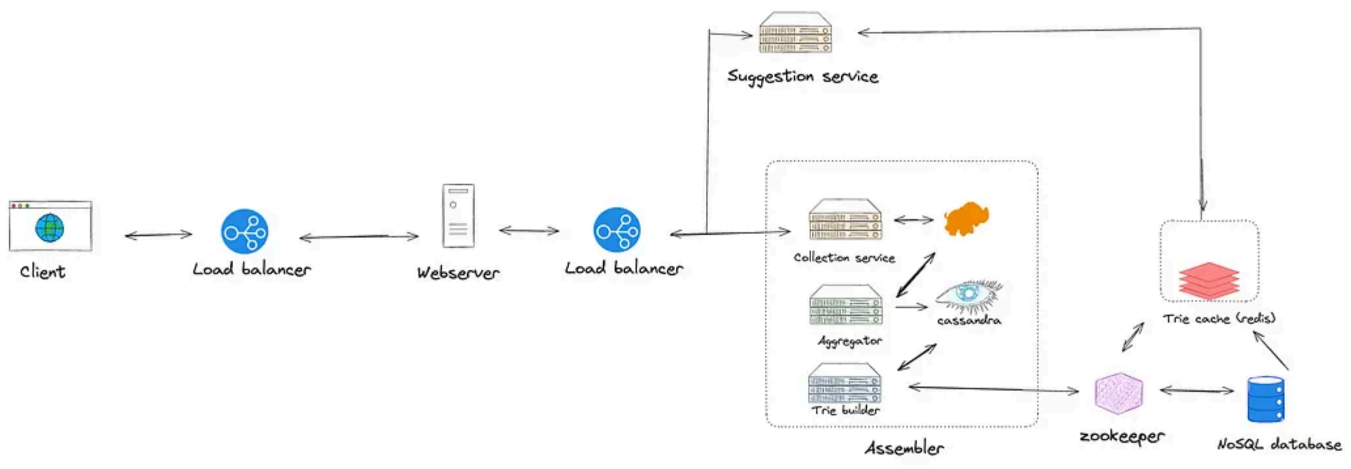


Fig 5.0: The detailed design of the typeahead suggestion system

Assembler

Creation and updation of a trie shouldn't come in the **critical path** of a user's query. We have a separate service called an assembler that's responsible for creating and updating tries after a certain configurable amount of time. The assembler consists of the following different services:

Collection service

Whenever a user types, this service collects the log that consists of phrases, time, and other metadata and dumps it in a database that's processed later. Since the size of this data is huge, the **Hadoop Distributed File System (HDFS)** is considered a suitable storage system for storing this raw data.

Aggregator

The raw data collected by the **collection service** is usually not in a consolidated shape. We need to consolidate the raw data to process it further and to create or update the tries. An aggregator retrieves the data from the HDFS and distributes it to different workers. Generally, the **MapReducer** is responsible for aggregating the frequency of the prefixes over a given interval of time, and the frequency is updated periodically in the associated Cassandra database. **Cassandra** is suitable for this purpose because it can store large amounts of data in a tabular format.

Trie builder

This service is responsible for creating or updating tries. It stores these new and updated tries on their respective shards in the trie database via **ZooKeeper**. Tries are stored in persistent storage in a file so that we can rebuild our trie easily if necessary. NoSQL document databases such as **MongoDB** are suitable for storing these tries. This storage of a trie is needed when a machine restarts.

The trie is updated from the aggregated data in the **Cassandra** database. The existing snapshot of a trie is updated with all the new terms and their corresponding frequencies. Otherwise, a new trie is created using the data in the Cassandra database.

Once a trie is created or updated, the system makes it available for the suggestion service.

Evaluation

Low latency

There are various levels at which we can minimize the system's latency. We can minimize the latency with the following options:

- Reduce the depth of the tree, which reduces the overall traversal time.
- Update the trie offline, which means that the time taken by the update operation isn't on the clients' critical path.
- Use geographically distributed application and database servers. This way, the service is provided near the user, which also reduces any communication delays and aids in reducing latency.
- Use Redis and Cassandra cache clusters on top of NoSQL database clusters.
- Appropriately partition tries, which leads to a proper distribution of the load and results in better performance.

Fault tolerance

Since the replication and partitioning of the trees are provided, the system operates with high resilience. If one server fails, others are on standby to deliver the services.

Scalability

Since our proposed system is flexible, more servers can be added or removed as the load increases. For example, if the number of queries increases, the number of partitions or shards of the trees is increased accordingly.

Summary

In this design problem, we learned how pushing resource-intensive processing to the offline infrastructure and using appropriate data structures enables us to serve our customers with low latency. Many optimizations lend themselves to specific use cases. We saw multiple optimizations on our trie data structures for condensed data storage and quick serving.

Typeahead Software Architect Software Architecture Distributed System Design System Design Interview

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium